Design Patterns and Principles

Created	@June 13, 2025 10:13 PM
Creted By - Sparsh	Ⓢ SPARSH SINGH Chundawat

SOLID Principle

source :- https://www.baeldung.com/solid-principles

https://medium.com/@softwaretechsolution/design-pattern-81ef65829de2

Also used chatGpt along with it.

Solid Principle is a part of OOD (Object-Oriented Design).

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

A. Single Responsibilty - a class should only have one responsibility.

Testing- A class with one responsibility will have fewer test cases.

Lower coupling - Less the functionality in a single class, fewer dependencies it will have.

Coupling means
how much one class is dependent on another. **Low coupling** = loosely connected = classes can work independently = more maintainable.

Organization – Smaller, well-organized classes are easier to search than monolithic ones.

example code:

```
// Responsible only for validation
class UserValidator {
public boolean isValid(String username, String email) {
return username != null && email != null;
}
}

// Responsible only for DB saving
class UserRepository {
public void save(String username) {
System.out.println("Saving " + username + " to the database");
}
}

// Responsible only for sending emails
class EmailService {
public void sendWelcomeEmail(String email) {
System.out.println("Sending welcome email to " + email);
}
}

// Main service class now has only 1 responsibility: user registration logic
class UserService {
private UserValidator validator = new UserValidator();
private UserRepository repository = new UserRepository();
private EmailService emailService = new EmailService();
public void registerUser(String username, String email) {
    if (!validator.isValid(username, email)) {
        System.out.println("Invalid input");
        return;
    }

    repository.save(username);
    emailService.sendWelcomeEmail(email);
}
}
// Testing it
public class Main {
public static void main(String[] args) {
UserService service = new UserService();
service.registerUser("sparsh", "sparsh@example.com");
//ab yeh service object- validate bhi kar dega, db me save bhi kardega and e
mail bhi bhejdega
}
}
```

Basically, yeh keh raha hai- har kaam ke liye alag class banao

B. Open for Extension , Closed for Modification -

Code ko aise likho ki agar naye feature add karne ho to existing code ko chhedna na pade. Sirf naye code likh ke kaam ho jaye.

i.e , **classes should be open for extension but closed for modification.**

```
public class Guitar {

    private String make;
    private String model;
    private int volume;
    //Constructors, getters & setters
}
```

But now we want to add Flame feature in guitar, so entering into class could welcome multiple bugs, so ...

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

C. Liskov Substitution-

Child class apne parent class ka "samman" kare. 😄

Agar koi class kisi parent class ko inherit karti hai, to usse parent ki jagah use karne par program sahi kaam kare — bina kisi unexpected behavior ke.

```
public interface Car {
void turnOnEngine();
void accelerate();
}

public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}

public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

```
Car myCar = new ElectricCar();
myCar.turnOnEngine(); // ❌ AssertionError: "I don't have an engine!"
```

You used a **child class** (`ElectricCar`) in place of a **parent type** (`Car`) — and the program broke.

That's **exactly what LSP says you should not do.**

Correct Design:-

```
//Interface for Car
public interface Car {
    void accelerate();
}
```

```
//Interface for EnginePowered because this was causing the issue
public interface EnginePowered {
    void turnOnEngine();
}
//MotoCar - runs fine
public class MotorCar implements Car, EnginePowered {
    private Engine engine;

    public void turnOnEngine() {
        engine.on();
    }

    public void accelerate() {
        engine.powerOn(1000);
    }
}
//ElectricCar - runs perfectly fine
public class ElectricCar implements Car {
    public void accelerate() {
        System.out.println("Zoom! Electric acceleration!");
    }
}
```

D. Interface Segregation-

larger interfaces should be split into smaller ones.

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

Ab bear ko na chahte hue bhi pet krna hi padega which is risky, so individual Interface banao

```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

Ab jab BearCarer aaega to usko pet krne ki jarurat nhi

```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //Bathing Time...
    }

    public void feedTheBear() {
        //Khana Khilade bhai...
    }
}
```

Koi pet krne ke liye aaega too easy hai uske liye bhi.

```
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

E. Dependecy Inversion-

Depemdecy Inversion that is Inverting the Dependencies or - **decoupling of software modules.**

"Bade log (high-level) chhoti details (low-level code) par dependent nahi hote, dono ek common rule (interface) follow karte hain.”

Code ko flexible banao – kisi cheez ko directly use mat karo, ek bridge (interface) se connect karo

```
class EmailService {
    public void sendEmail(String message) {
        System.out.println("Sending email: " + message);
    }
}

class Notification {
    private EmailService emailService = new EmailService(); // tightly coupled

    public void alert(String message) {
        emailService.sendEmail(message);  // hardcoded
    }
}
```

But this is the correct code:-

```
//Creating an Interface using Abstraction
interface MessageService {
    void send(String message);
}
```

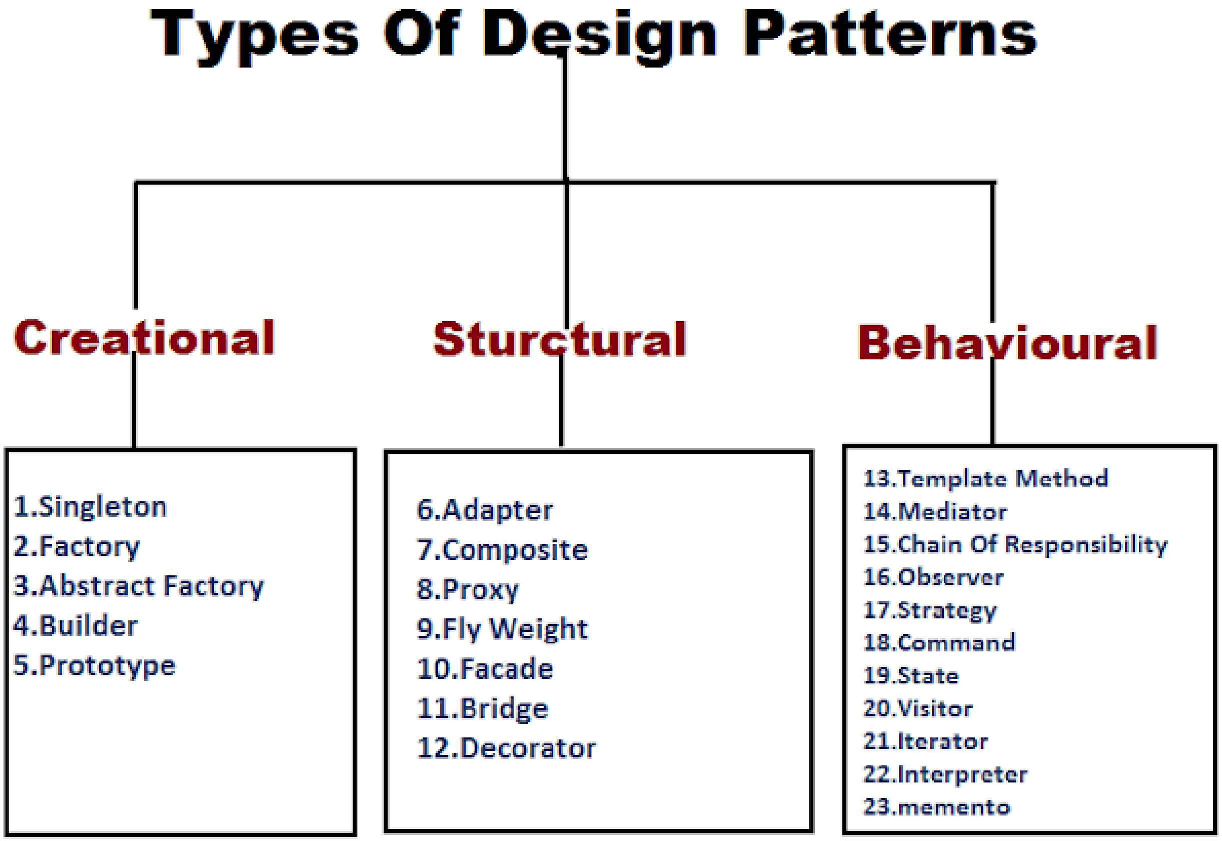
```
//Create concrete implementations
class EmailService implements MessageService {
    public void send(String message) {
        System.out.println("📧 Email: " + message);
    }
}

class SMSService implements MessageService {
    public void send(String message) {
        System.out.println("📱 SMS: " + message);
    }
}
//High Level class depends on Abstraction
class Notification {
    private MessageService service; // interface

    // Constructor Injection
    public Notification(MessageService service) {
        this.service = service;
    }

    public void alert(String message) {
        service.send(message); // No idea who’s sending — flexible!
    }
}
```

Design Patterns:

There are majorly three types of Design Patterns

1. **Creational Patterns**
2. Structural Patterns

3. Behavioral Patterns



The Decorator pattern is also known as Wrapper because it's used to wrap an object to add new behavior to it.